Mejoras e Implementación del Algoritmo Marching Cubes

Alejandro Kastner
akastner@dc.uba.ar

Camilo Melani
cmelani@dc.uba.ar

Director
MS. Claudio Delrieux

Tesis de Licenciatura

Noviembre 2004

Resumen

En este trabajo se presenta una solución a los problemas de generación de huecos que tienen los algoritmos tradicionales que generan isosuperficies. También se implementa una aplicación capaz de visualizar datos médicos volumétricos. Esto se realiza con técnicas de visualización de múltiples isosuperficies, generación de planos, *clipping* y transparencia.

Agradecimientos

A Claudio por su colaboración, su esfuerzo y su ayuda. Fuimos muy afortunados en haberlo conocido, y que nos aceptara como sus tesistas.

A Andrea Silvetti por colaborar, comprometerse, explicarnos, y aclararnos nuestras dudas en todo momento.

Al Dr Juan Mazzuco por su paciencia y desinteresada ayuda.

A todos nuestros compañeros de estudio por ayudarnos a lo largo de toda la carrera.

A los compañeros de trabajo, por escucharnos y compartir tantas horas con nosotros.

A nuestros padres que sin ellos hubiera sido imposible llegar hasta aquí.

A familiares, amigos y novias.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Las superficies anatómicas tridimensionales son una herramienta de gran valor para la medicina. Las imágenes de tales superficies, construidas a partir de múltiples imágenes paralelas de dos dimensiones obtenidas por tomografías computadas o resonancias magnéticas (ver Figura 1.1), ayudan a los médicos a entender la compleja anatomía. La visualización de complejas fracturas, anomalías cráneo-faciales y la estructura intercraneal ilustran el potencial de contar con superficies tridimensionales para el estudio de estructuras óseas complejas.

Pero no sólo en el área médica es útil trabajar con superficies en tres dimensiones. La posibilidad de generar superficies tridimensionales a partir de datos volumétricos obtenidos de fuentes diversas como ser funciones o mediciones con sensores nos muestra la diversidad de disciplinas en las cuales visualizar isosuperficies sirve como una herramienta más de análisis de la información.

Existen numerosos enfoques al problema de la generación de superficies tridimensionales. Una técnica primaria analiza los contornos de la superficie a construir y conecta los contornos de las imágenes consecutivas con triángulos [Wat92]. Desafortunadamente, si existe más de un contorno en una imagen se generan ambigüedades al momento de determinar cuáles contornos conectar. La intervención interactiva del usuario puede resolver algunas de dichas ambigüedades; sin embargo, en un ambiente clínico la interacción del usuario debe mantenerse al mínimo. Otro enfoque es utilizar *ray casting* [Wat92] para visualizar la superficie tridimensional. Para ello se calcula un gradiente para los datos escalares y luego se utiliza este gradiente, escalado por un valor apropiado, para generar un sombreado en escala de grises de la imagen por medio de algoritmos similares al *ray tracing* [Fol90]. Cada una de

Figura 1.1: Conjunto de tomografías.

las técnicas de construcción de superficies mencionadas presenta defectos debido a que desechan información importante de los datos adquiridos, generan dificultades operativas en el manejo, o son de un costo computacional excesivo, lo cual excluye la posibilidad de realizar visualizaciones interactivas del volumen de datos.

El enfoque que desarrollamos en este trabajo, basado en el algoritmo *marching cubes* [Lor87] es más simple y rápido. Este algoritmo utiliza la información de los datos adquiridos para deducir la conectividad, la ubicación y el gradiente de la isosuperficie. Con dichos datos realiza una poligonización que ajusta por primer orden a la superficie umbral que rodea a los datos de interés. El modelo de triángulos resultante puede ser fácilmente graficado con algoritmos estándar o bibliotecas de computación gráfica.

Este algoritmo en la versión original presenta inconvenientes para ciertos casos, generando superficies con huecos que no representan fielmente a la superficie que se intenta modelar [Dur88].

En las siguientes secciones analizamos los problemas que presenta este algoritmo, introducimos mejoras e implementamos una aplicación que puede ser utilizada fácilmente por usuarios que requieren este tipo de herramientas.

Capítulo 2

Proceso completo para la obtención de superficies

Las sistemas de visualización de imágenes médicas tridimensionales constan de cuatro pasos (ver Figura 2.1):

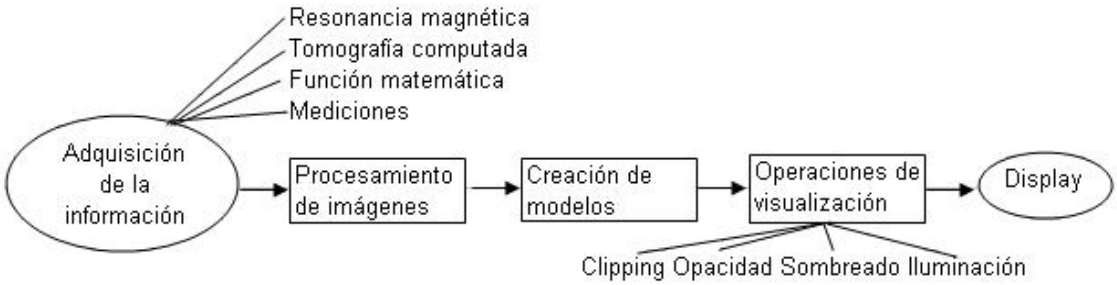


Figura 2.1: Etapas necesarias para la construcción de superficies.

2.1. Adquisición de la información

En el caso de la medicina este paso es realizado por un instrumento médico de imágenes, el cual obtiene información de un paciente y produce múltiples fetas. La información recolectada proviene de una tomografía computada o bien de una resonancia magnética. Otras fuentes de información pueden ser microscopios confocales, funciones matemáticas o dispositivos que realizan algún tipo de medición (ver Figura 2.2), como por ejemplo sensores de temperatura o presión.

Figura 2.2: Tomógrafo.

2.2. Segmentación de imágenes

Consiste en distinguir en la imagen la parte que nos interesa representar en tres dimensiones. Algunos algoritmos usan técnicas de procesamiento de imágenes [Gon96], como por ejemplo la utilización de filtros que ayudan a encontrar características de forma dentro de la imagen (ver Figura 2.3).

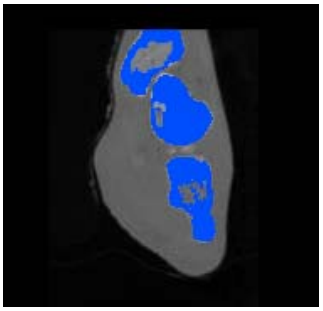


Figura 2.3: Tomografía axial del pie. El color azul marca la zona a reconstruir.

2.3. Construcción de la superficie

Este paso involucra la generación de los triángulos que forman la superficie a partir de los datos volumétricos. La técnica que estudiamos en este trabajo se denomina *marching cubes* (ver Figura 2.4).

2.4. Visualización

Realizar el renderizado de la superficie usando los modelos de computación gráfica. Para analizar las superficies obtenidas se pueden combinar distintas técnicas de acuerdo a la necesidad del usuario, por ejemplo manipulación interactiva del objeto

Figura 2.4: Reconstrucción del pie con tres aumentos distintos.

en tres dimensiones, sombreado, opacidad, *clipping*, cortes, iluminación, renderización de múltiples superficies simultáneamente (ver Figura 2.5).

Figura 2.5: Visualización de dos superficies generadas a partir del mismo conjunto de datos pero reconstruidas con distintas opacidades.

Capítulo 3

Algoritmo Marching Cubes

3.1. Introducción

El algoritmo procesa la grilla cúbica de datos celda a celda en un orden determinado produciendo una isosuperficie de manera local. Una celda en el espacio está delimitada por los ocho valores de sus vértices y se clasifica según los valores de sus vértices respecto al valor umbral. Cada uno de los ocho vértices de una celda puede asumir un valor superior (marcado) o inferior (no marcado), por lo tanto una celda pertenece a la isosuperficie si por lo menos uno de sus vértices está por debajo del valor umbral y por lo menos otro está por encima. Al haber 8 vértices en cada celda y 2 estados (marcado o no) existen $2^8 = 256$ maneras que una superficie puede intersectar a una celda. Lorensen y Cline [Lor87] proponen 15 formas básicas (ver Figura 3.1), tomando los casos restantes como complemento y/o rotaciones de las primeras. Tener todos los casos resumidos en 15 clases de equivalencias sirve para reducir el análisis de cada celda. Como cada grupo genera las mismas triangulaciones, sólo basta rotar los vértices de los triángulos para obtener la superficie que corresponde.

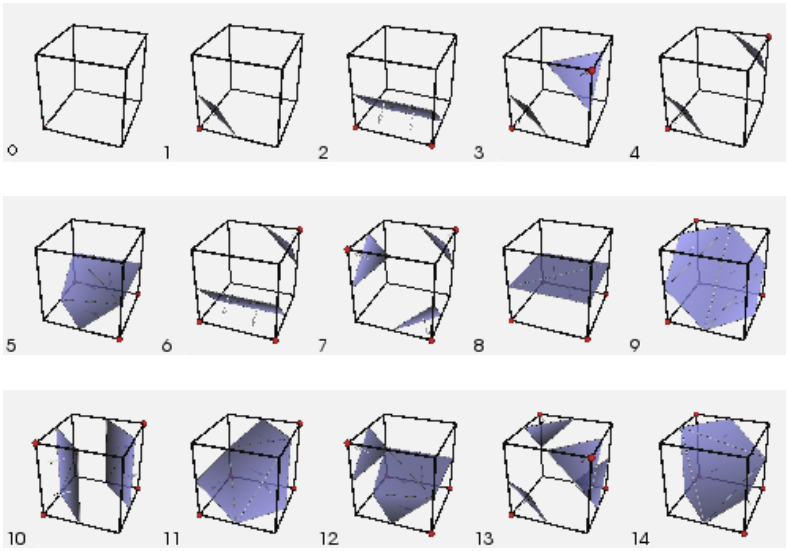


Figura 3.1: Los 15 casos propuestos por Lorensen y Cline.

3.2. Problemas encontrados

El algoritmo *Marching Cubes* basado en las triangulaciones planteadas por [Lor87] conduce a la formación de superficies no deseadas (huecos). Esto es producto de que las configuraciones propuestas son incompletas y surgen ambigüedades en la elección de la triangulación que mejor representa a la isosuperficie que atraviesa a la celda.

La ambigüedad se produce cuando la cara que comparten dos celdas adyacentes tiene marcado sólo dos vértices en forma diagonal (ver Figura 3.2).

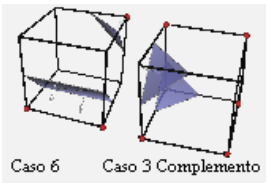


Figura 3.2: Problemas encontrados.

Cuando dos celdas comparten una cara ambigua y los puntos con valor superior al umbral de esta cara quedan separados en una celda y en la otra quedan unidos se generan huecos en la isosuperficie. Esto ocurre cuando una de las celdas tiene a lo sumo cuatro puntos marcados mientras que la otra tiene al menos cuatro, entonces una de ellas debe invertir los puntos marcados (complemento) antes de ser procesada y luego invertir las normales de los triángulos generados para esa celda. En este punto se ve la necesidad de distinguir entre una cara ambigua separando los puntos marcados y una cara ambigua uniendo dichos puntos, es decir, hay dos posibles conexiones para aparear los cuatro puntos que forman los triángulos de esa configuración.

Para lograr una superficie topológicamente correcta, las dos celdas en cuestión deben optar por la misma conexión y en función de esta decisión elegir la triangulación correcta en cada caso.

3.3. Clasificación de las caras ambiguas.

Para resolver esta ambigüedad en [Nie91] se propone un método para elegir correctamente qué vértices conectar para que no se generen caras ambiguas.

Dados los valores de los vértices de la cara ambigua se puede construir una función bivariada que toma valores en toda la superficie de la cara interpolando los vértices.

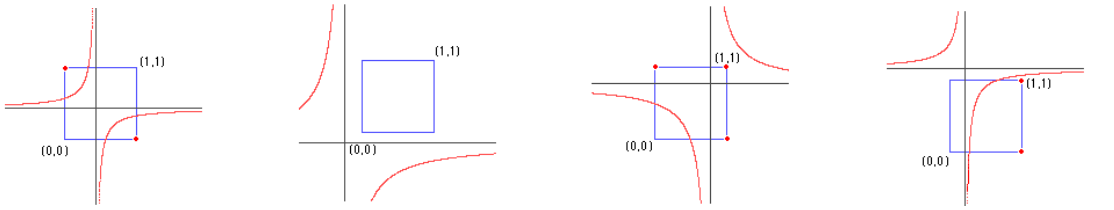
Considerando el tamaño de la cara igual a la unidad se obtiene

$$B(s,t) = \begin{pmatrix} 1-s, & s \end{pmatrix} \begin{pmatrix} b_{00} & b_{10} \\ b_{01} & b_{11} \end{pmatrix} \begin{pmatrix} 1-t \\ t \end{pmatrix} \tag{3.1}$$

donde $0 \leqslant s,t \leqslant 1$ y b_{00}, b_{01}, b_{10}y b_{11} son los valores de los vértices de la cara ambigua.

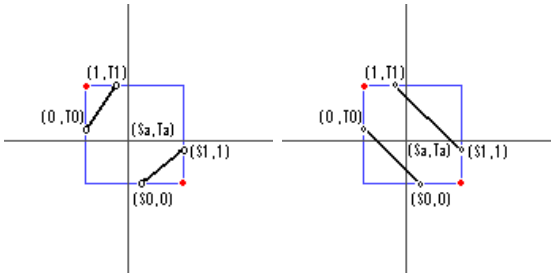
La función

$$B(s,t) = \alpha \tag{3.2}$$



(a) Cara ambigua (b) La isosuperficie no corta la cara (c) Cara no ambigua (d) Cara no ambigua

Figura 3.3: Análisis de las caras.



(a) Cara separada (b) Cara unida

Figura 3.4: Tipo de cara.

donde α es el valor de la isosuperficie que se intenta modelar, forma una hipérbola

El caso ambiguo surge cuando ambos componentes de la hipérbola cortan el dominio de B (ver Figura 3.3). La ambigüedad se resuelve aplicando el siguiente test (*asymptotic decider*):

$$\begin{cases} Conectar\ (S_1,1)\,a\,(1,T_1) \\ \quad y \quad (S_0,0)\,a\,(0,T_0) \end{cases} \quad Si\,\alpha > B(s_a,t_a)\ \ Cara\,unida$$
$$\begin{cases} Conectar\ (S_1,1)\,a\,(S_0,0) \\ \quad y \quad (0,T_0)\,a\,(1,T_1) \end{cases} \quad Si\,\alpha \leq B(s_a,t_a)\ \ Cara\,separada \tag{3.3}$$

donde (s_a, t_a)es el punto asintótico.

Si el resultado de aplicar (3.3) a una cara indica conexión entre dos aristas que comparten un punto mayor al isovalor entonces se dice que esa cara es separada, en otro caso es unida (ver Figura 3.4).

Una vez identificadas las caras ambiguas, para evitar los huecos se eligen las triangulaciones para las dos celdas que comparten la cara ambigua de tal forma que ambas celdas tengan a esa cara separada o unida.

En [Nie91] se introducen nuevas triangulaciones al algoritmo que tienen en cuenta si la cara ambigua es separada o unida, pero aún así resultan incompletas ya que no se analizan exhaustivamente todas las combinaciones de casos con caras ambiguas.

3.4. Triangulaciones que no generan huecos

El trabajo de Silvetti [Sil01] corrige los problemas encontrados en la solución propuesta por Nielson y Hamann, y describe las triangulaciones que es necesario agregar a las ya propuestas.

Existen casos en que es necesario, además de las seis caras que delimitan a la celda, considerar una cara extra diagonal (ver Figura 3.5) para evitar obtener superficies no esperadas. También es preciso distinguir entre caras ambiguas orientadas hacia la derecha y caras ambiguas orientadas hacia la izquierda (ver Figura 3.6).

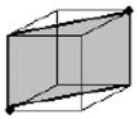


Figura 3.5: Cara diagonal.

(a) Separada a derecha

(b) Separada a izquierda

(c) Unida a derecha

(d) Unida a izquierda

Figura 3.6: Tipos de cara.

Analizando las 15 configuraciones propuestas por Lorensen y Cline (ver Figura 3.1), las triangulaciones 1, 2, 5, 8, 9, 11, 14 no tienen caras ambiguas, por lo tanto las triangulaciones se mantienen tal cual las originales. Los casos restantes presentan caras ambiguas y es necesario que sean tratados individualmente.

3.4.1. Casos con una cara ambigua

Las triangulaciones 3, 4 y 6 presentan una cara ambigua y se resuelven aplicando el método del *asymptotic decider*. En el caso 3 la cara ambigua es la del frente, en el caso 3A la consideramos como separada a derecha mientras que en el 3B unida a derecha (ver Figura 3.7).

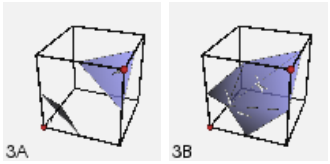


Figura 3.7: Caso 3A y 3B.

En el caso 4 la cara ambigua es la diagonal, en el caso 4A la consideramos como separada a derecha mientras que en el 4B unida a derecha (ver Figura 3.8).

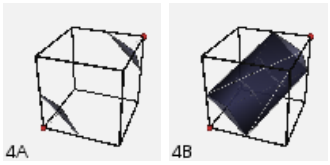


Figura 3.8: Caso 4A y 4B.

Nielson y Hamann no tienen en cuenta la ambigüedad de la cara diagonal y, si bien no se generan huecos por la elección de una u otra configuración, separar este caso en dos genera superficies que representan de mejor forma la realidad.

En el caso 6 la cara ambigua es la derecha, en el caso 6A la consideramos como separada a derecha mientras que en el 6B unida a derecha (caso similar al 3) (ver Figura 3.9).

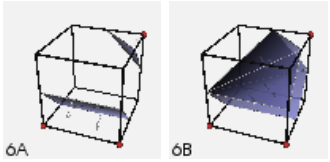


Figura 3.9: Caso 6A y 6B.

3.4.2. Casos con dos caras ambiguas

El caso 10 presenta dos caras ambiguas, la superior e inferior (ver Figura 3.10)

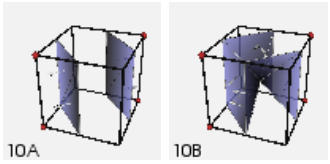


Figura 3.10: Caso 10A y 10B.

En este caso se presentan cuatro combinaciones posibles, pero mediante transformaciones convenientemente aplicadas (rotaciones y/o complementos) sólo bastan dos triangulaciones.

- Superior separada a derecha e inferior separada a izquierda. Caso 10A.
- Superior unida a izquierda e inferior unida a derecha. Caso 10A Complemento.
- Superior separada a derecha e inferior unida a izquierda. Caso 10B.
- Superior unida a izquierda e inferior separada a derecha. Caso 10B Complemento.

El caso 12 presenta dos caras ambiguas, la izquierda y la del frente (ver Figura 3.11)

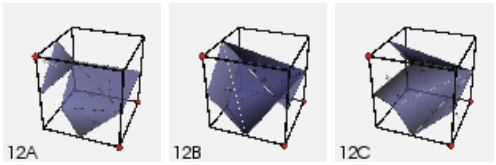


Figura 3.11: Caso 12A, 12B y 12C.

En este caso se presentan cuatro combinaciones posibles, pero mediante transformaciones convenientemente aplicadas (rotaciones y/o complementos) sólo bastan tres triangulaciones.

- Izquierda separada a derecha y frente separada a izquierda. Caso 12A.
- Izquierda unida a izquierda y frente unida a derecha. Caso 12A Complemento.
- Izquierda separada a derecha y frente unida a izquierda. Caso 12B.
- Izquierda unida a derecha y frente separada a izquierda. Caso 12C.

3.4.3. Casos con tres caras ambiguas

El caso 7 presenta tres caras ambiguas, la superior, la derecha y la del frente. Para lograr las triangulaciones adecuadas para este caso, es necesario generar triángulos que tengan vértices en el centro de la celda (en las configuraciones de Lorensen y Cline esto no se tiene en cuenta) (ver Figura 3.12).

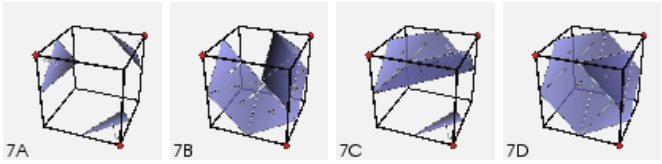


Figura 3.12: Caso 7A, 7B, 7C y 7D.

En este caso se presentan 8 combinaciones posibles, pero mediante transformaciones convenientemente aplicadas (rotaciones y/o complementos) sólo bastan cuatro triangulaciones.

- Superior separada a derecha, la derecha separada a derecha y la del frente separada a izquierda. Caso 7A.
- Superior unida a izquierda, la derecha unida a izquierda y la del frente unida a derecha. Caso 7A Complemento.
- Superior separada a derecha, la derecha unida a derecha y la del frente unida a izquierda. Caso 7B.
- Superior unida a izquierda, la derecha separada a izquierda y la del frente separada a derecha. Caso 7B Complemento.
- Superior unida a derecha, la derecha separada a derecha y la del frente separada a izquierda. Caso 7C.
- Superior separada a izquierda, la derecha unida a izquierda y la del frente unida a derecha. Caso 7C Complemento.
- Superior unida a derecha, la derecha unida a derecha y la del frente unida a izquierda. Caso 7D.
- Superior separada a derecha, la derecha separada a derecha y la del frente separada a izquierda. Caso 7D Complemento.

3.4.4. Casos con seis caras ambiguas

El caso 13 tiene todas sus caras ambiguas (ver Figura 3.13). Cuatro orientadas hacia un lado y dos hacia el otro.

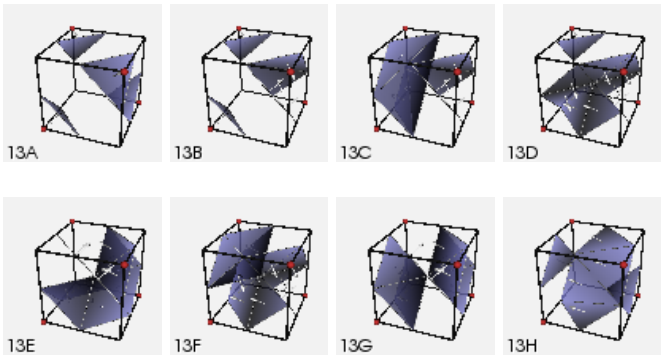


Figura 3.13: Caso 13.

Si la cantidad de caras separadas es 6 se elige el caso 13A, si no tiene ninguna cara separada la triangulación se resuelve usando el complemento del caso 13A rotando e invirtiendo las normales según corresponda.

De modo similar se resuelven los casos con 5 o 1 caras separadas, eligiendo el caso 13B y 13B Complemento.

Cuando la cantidad de caras separadas es 2 o 4, es necesario distinguir la relación entre las dos caras con igual conexión (las dos separadas o las dos unidas). De ser opuestas optamos por la configuración 13C, mientras que si son adyacentes se debe elegir la configuración 13D. Sólo en el caso en el que haya dos caras separadas es necesario invertir las normales (además de las rotaciones adecuadas).

Cuando se tienen tres caras separadas y tres unidas, se deben considerar cuidadosamente todas las combinaciones posibles, ya que existen cuatro caras orientadas hacia la izquierda y dos hacia la derecha mientras que en el complemento hay cuatro hacia la derecha y dos hacia la izquierda. Entonces es necesario distinguir entre los casos en que las tres caras se unen en un punto marcado o en un punto no marcado, o en una cara ambigua hacia la derecha o en una cara ambigua hacia la izquierda. Si las 3 caras concurren a un punto, se elige la triangulación 13H si el punto concurrente es superior al umbral (marcado), de no ser así, se elige el 13G. Cuando las 3 caras no concurren en un punto se tiene dos caras opuestas. Si la restante está orientada a la izquierda la triangulación apropiada es la 13E, si no es la 13H.

Capítulo 4

Implementación del algoritmo

4.1. Introducción

Para implementar este algoritmo utilizamos la biblioteca VTK (ver Capítulo 6), una biblioteca *open source* desarrollada en clases de C++ orientada a visualización científica. VTK cuenta con clases que actúan como filtros, es decir transforman mediante un método un conjunto de datos de entrada generando como salida otro conjunto de datos, el cual a su vez puede ser la entrada de datos de otra clase.

Para *marching cubes,* esta biblioteca implementa una variante que utiliza las triangulaciones propuestas por Nielson y Hamann.

El objetivo de crear otra versión del algoritmo *marching cubes* no sólo consiste en obtener una superficie que represente de mejor forma la realidad, sino que también pueda ser utilizado de igual manera que la versión implementada en la biblioteca. Esto significa que creamos una clase que tiene la misma entrada y salida de datos que la versión original, proporcionándole al usuario de VTK una alternativa mejorada de *marching cubes* que no requiere de un tratamiento distinto. Por lo tanto, cualquier conjunto de datos que es usado de entrada por el algoritmo de la biblioteca puede ser usado como entrada por nuestro algoritmo, mientras que los filtros aplicables al conjunto de datos de salida del algoritmo original pueden ser aplicados al nuestro. Es por esto que la clase que implementa nuestra versión del algoritmo hereda, en la estructura de clases de VTK, de la clase padre de la clase que implementa el *marching cubes*.

4.2. Desarrollo

Para lograr que este algoritmo funcione eficientemente contamos con varias estructuras de datos que facilitan la rápida y correcta identificación del caso a analizar, y a partir del mismo determinar qué triángulos generar. Por ejemplo, para cada caso que presenta alguna ambigüedad existe una matriz que indica la transformación a realizar dada la configuración de cada cara (unida o separada a derecha o a izquierda). También es necesario contar con estructuras que indiquen qué triángulos genera cada caso, cuáles son las rotaciones que se deben aplicar y las clases de equivalencia de cada caso.

Al desarrollar nuestra versión del algoritmo *marching cubes* creamos una clase

denominada *vtkMarchingCubesTesis* que tiene la misma clase padre que la clase original, de modo de poder usar indistintamente una u otra.

En el árbol de clases de VTK (ver Apéndice, Figura 10.1), *marching cubes* hereda de la clase *vtkStructuredPointsToPolyDataFilter*. Esto determina que los objetos de esta clase actúan como filtros, recibiendo como entrada un objeto de la clase *vtkImageData* (clase padre de *vtkStructuredPoints*) para devolver un objeto de la clase *vtkPolyData*.

Al ser un algoritmo iterativo que genera isosuperficies de manera local, es necesario conocer los ocho puntos de la celda que estamos procesando. Como la información dentro de la clase *vtkImageData* se encuentra almacenada en un arreglo, para acceder a la posición que nos interesa debemos indexar teniendo en cuenta las dimensiones del volumen.

Por ejemplo, si procesamos un volumen de dimensión 3x2x2 y queremos generar los triángulos para la celda (x,y,z)=(1,0,0) (ver Figura 4.1), buscamos cada uno de sus vértices de la siguiente manera:

$$Vertice0 = x + y * dim_x + (z + 1) * dim_y * dim_z$$

$$Vertice1 = (x + 1) + y * dim_x + (z + 1) * dim_y * dim_z$$

$$Vertice2 = (x + 1) + y * dim_x + z * dim_y * dim_z$$

$$Vertice3 = x + y * dim_x + z * dim_y * dim_z$$

$$Vertice4 = x + (y + 1) * dim_x + (z + 1) * dim_y * dim_z$$

$$Vertice5 = (x + 1) + (y + 1) * dim_x + (z + 1) * dim_y * dim_z$$

$$Vertice6 = (x + 1) + (y + 1) * dim_x + z * dim_y * dim_z$$

$$Vertice7 = x + (y + 1) * dim_x + z * dim_y * dim_z$$

donde dim_i= Dimensión i

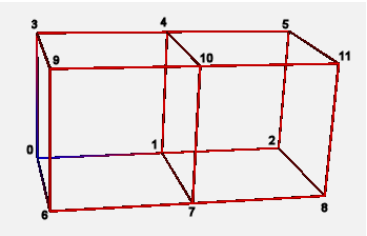


Figura 4.1: Grilla 3x2x2.

En la figura 4.1 la celda de la derecha corresponde a la celda (1,0,0).

Una vez obtenidos los valores de los vértices pertenecientes a una celda conocemos su topología, es decir, cuáles vértices están marcados. A partir de esto obtengo un

identificador de caso, ya que cada una de las 256 combinaciones posibles tiene un identificador propio (ver Figura 4.2).

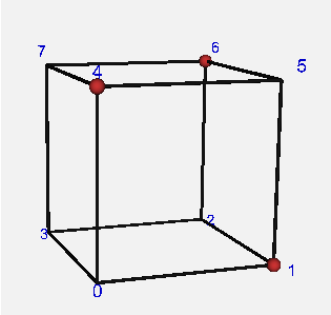


Figura 4.2: Vértices marcados: 1, 4 y 6. $Identificador = 2^1 + 2^4 + 2^6 = 82$.

Como los casos que analizamos son los que tienen a lo sumo 4 puntos marcados, transformamos los que tienen 5 o más en su complemento. Esto es posible porque la triangulación resultante es la misma, salvo por la orientación de las normales de cada triángulo. Para esto contamos con una estructura de datos que representa a un conjunto donde almacenamos los casos que tienen más de 4 puntos marcados. Si el caso en análisis pertenece a este conjunto, entonces lo transformamos en su complemento al restarle a 255 el identificador de caso.

Dado que existen muchas configuraciones que generan las mismas triangulaciones (por ejemplo, las rotaciones de un caso) basta con agrupar en clases de equivalencia para tratar a todos sus miembros de la misma forma. En total, son 14 grupos. Para obtener el grupo al cual pertenece el caso que se analiza, contamos con una estructura de datos que representa un diccionario que tiene como clave el identificador de caso y como definición el grupo al cual pertenece.

También es importante determinar en qué posición analizar un determinado caso. La posición indica la rotación que tiene aplicada una celda. Para esto contamos con otro diccionario que almacena como definición la posición de análisis y como clave de búsqueda al identificador de caso.

Luego, pasamos a usar la función que determina qué triangulación realizar y qué rotación aplicarle. Esta función usa una tabla que contiene la información de qué triángulo generar para cada caso (ver Cuadro 10.14).

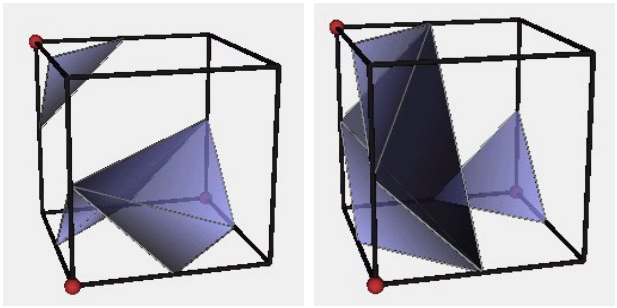
Ahora analizamos caso por caso qué decisión tomar:

- Si la clase de equivalencia del caso es 1, 2, 5, 8, 9, 11 o 14, o sea, casos que no tienen caras ambiguas, la triangulación se deduce inmediatamente.

En caso de que la clase de equivalencia tenga caras ambiguas es necesario analizar más detalladamente la situación. Por empezar, se determina el tipo de cada cara ambigua, es decir, si es separada o unida mediante el algoritmo *Asymptotic Decider*. La información sobre el tipo de cada cara se almacena en un arreglo, el cual se recalcula en cada iteración. En el caso en que la configuración en análisis provenga de un caso perteneciente a los complementos, entonces el tipo de cara se invierte, o sea, si es unida pasa a ser separada y viceversa.

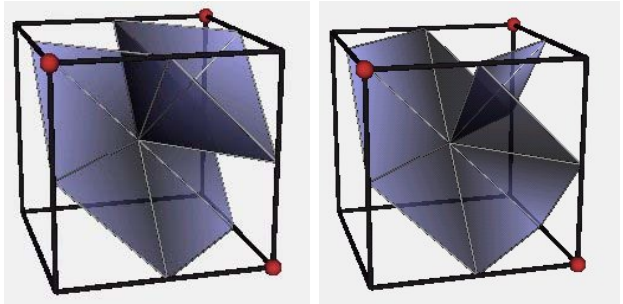
- Si la clase de equivalencia es 3, 4 o 6 (casos con sólo una cara ambigua), se determina primero cuál es la cara ambigua usando la función *"CaraARevisar"*[1] y luego si ésta es separada o unida. De ser separada, la triangulación elegida es la A, caso contrario la B (3A, 4A, 6A o 3B, 4B, 6B respectivamente).

- Si la clase de equivalencia es la 7, se determina cuáles son las tres caras ambiguas usando la función *"CarasAContar"*[2] . Luego, se determina cuántas de ellas son separadas. De ser las tres o ninguna separadas, se eligen las triangulaciones 7A y 7D respectivamente sin modificar la posición de análisis. Si hay una o dos caras separadas (triangulaciones 7B y 7C), es necesario modificar la posición de análisis de acuerdo a la posición que se tiene (ver Figura 4.3,4.4). Para esto se utiliza un diccionario, que dada la posición de análisis actual devuelve la rotación a aplicar (ver Cuadro 10.1).



(a) Posición entrada: 11 (b) Posición nueva: 14

Figura 4.3: Triangulación 7C. Caras separadas: Inferior y Fondo.



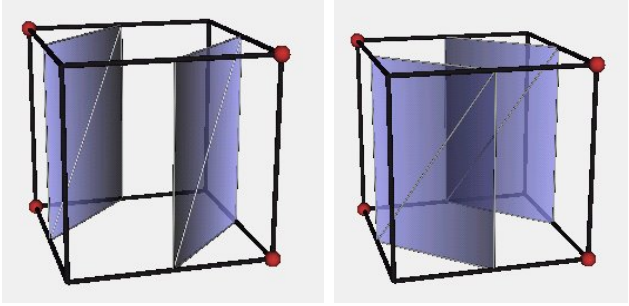
(a) Posición entrada: 1 (b) Posición nueva: 8

Figura 4.4: Triangulación 7B. Cara separada: Superior.

[1] CaraARevisar: Esta función devuelve la cara ambigua de una clase de equivalencia que tenga una sola cara ambigua (3, 4, 6). Para esto se vale de la información de qué puntos están marcados en la celda.

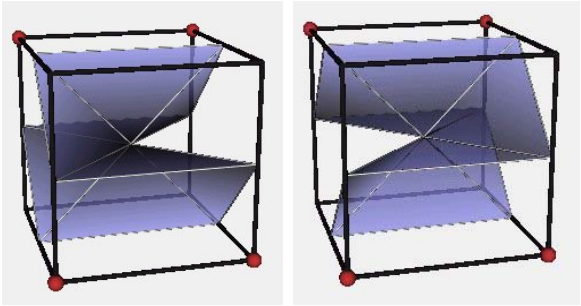
[2] CarasAContar: Esta función devuelve las caras ambiguas para las clases de equivalencia 7, 10 y 11. Para esto se vale de la información de qué puntos están marcados en la celda.

- Si la clase de equivalencia es la 10, se determina cuáles son las dos caras ambiguas usando la función *"CarasAContar"*. Luego, si las dos o ninguna son separadas, se elige la triangulación 10A, pero en el segundo caso se invierten las normales (ver Figura 4.5). De haber una separada y una unida se usa la triangulación 10B y aparte se modifica la posición de análisis de acuerdo a la posición que se tiene (ver Figura 4.6), para lo cual también se usa un diccionario (ver Cuadro 10.2).



(a) Posición entrada: 7 (b) Posición nueva: 1

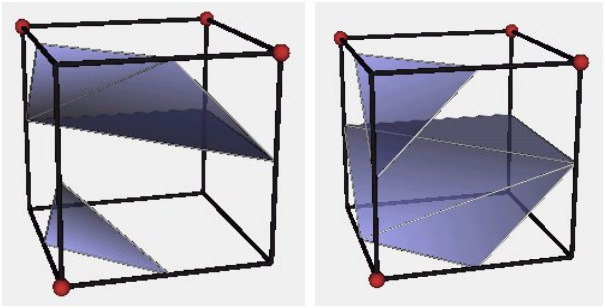
Figura 4.5: Triangulación 10A. Cara separada: Superior.



(a) Posición entrada: 10 (b) Posición nueva: 16

Figura 4.6: Triangulación 10B. Cara separada: Izquierda.

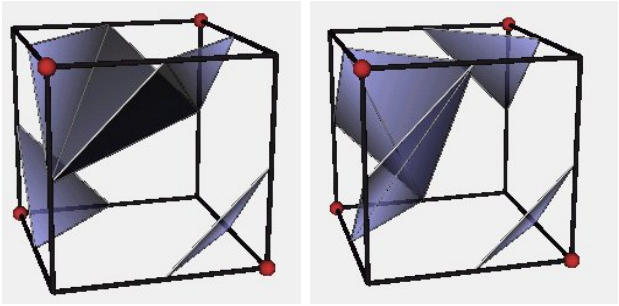
- Si la clase de equivalencia es la 12, se determina cuáles son las dos caras ambiguas usando la función *"CarasAContar"*. Si las dos caras son separadas, se elige la triangulación 12A. En el caso en que haya una sola cara separada, es necesario distinguir si ésta es la cara izquierda suponiendo que la posición de análisis es la 1 (cuando la posición no es la 1, es necesario buscar la cara que, luego de rotada, se transforma en la izquierda). En este caso se elige la triangulación 12B, si no la 12C. La última posibilidad es que ninguna de las dos caras ambiguas sea separada. En este caso se elige también la triangulación 12A, pero se invierten las normales y se modifica la posición de análisis de acuerdo a la posición que se tiene (ver Figura 4.7), para lo cual también se usa un diccionario (ver Cuadro 10.3).



(a) Posición entrada: 11 (b) Posición nueva: 1

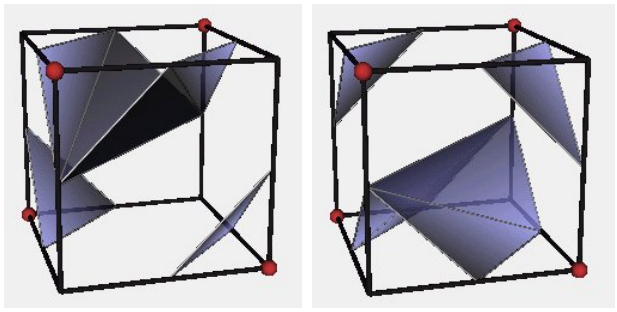
Figura 4.7: Triangulación 12A. Ninguna cara separada.

- Si la clase de equivalencia es la 13 (todas sus caras son ambiguas), se determina cuántas de sus caras son separadas. Cuando todas o ninguna son separadas, la triangulación que se escoge es la 13A, con la diferencia que en el segundo caso se invierten las normales. Cuando son cinco o una las caras separadas, la triangulación elegida es la 13B. Es necesario modificar la posición teniendo en cuenta la posición de análisis, la cantidad de caras separadas y el tipo de conexión de cada cara. Si la posición es 2 y hay 5 caras separadas o la posición es 1 y hay una cara separada se usa un diccionario (ver Cuadro 10.4 y Figura 4.8) y en el caso contrario otro (ver Cuadro 10.5 y Figura 4.9). Además, se deben invertir las normales si tiene una sola cara separada.



(a) Posición entrada: 2 (b) Posición nueva: 21

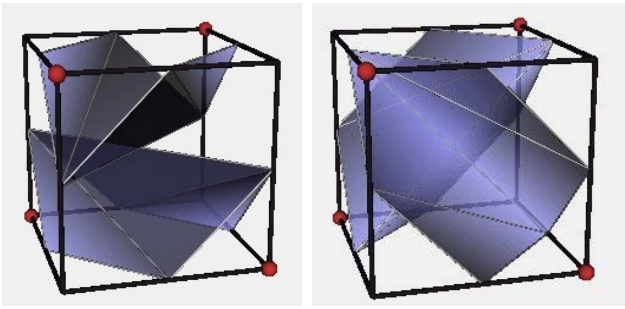
Figura 4.8: Triangulación 13B .Todas las caras separadas salvo la izquierda.



(a) Posición entrada: 2 (b) Posición nueva: 22

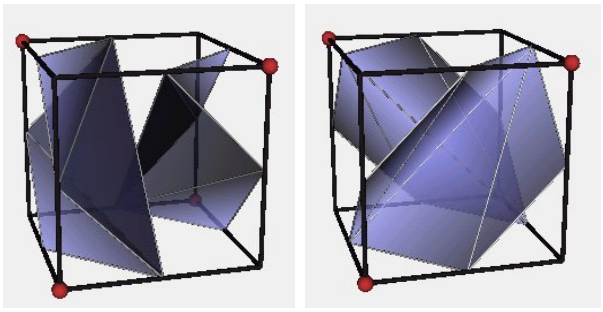
Figura 4.9: Triangulación 13B. Cara separada: Inferior.

- En el caso en que haya dos o cuatro caras separadas, se debe analizar si las dos caras con igual conexión son opuestas o adyacentes (las dos separadas o las dos unidas respectivamente). Si son opuestas, se elige la triangulación 13C y se modifica la posición de análisis de acuerdo al tipo de conexión de cada cara. Para esto se usa un diccionario (ver Cuadro 10.6) cuando son dos las caras separadas y la posición es 1 o hay cuatro caras separadas y la posición es 2 (ver Figura 4.10). En el caso contrario se usa otro diccionario (ver Cuadro 10.7 y Figura 4.11) y se invierten las normales. Si son adyacentes las caras, se elige la triangulación 13D y se modifica la posición de análisis de acuerdo al tipo de conexión de cada cara. Para esto se usa un diccionario (ver Cuadro 10.8) cuando son dos las caras separadas y la posición es 1 o hay cuatro caras separadas y la posición es 2 (ver Figura 4.12). Para el caso contrario se usa otro diccionario (ver Cuadro 10.9 y Figura 4.13) y se invierten las normales.



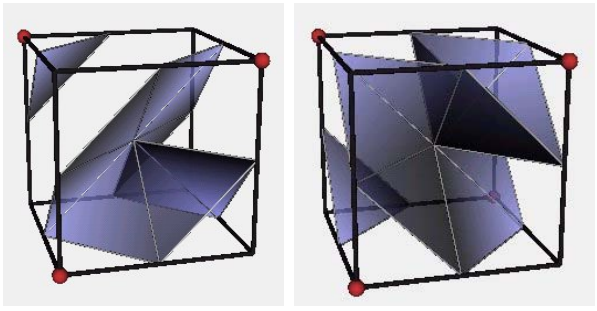
(a) Posición entrada: 2 (b) Posición nueva: 11

Figura 4.10: Triangulación 13C. Caras Separadas: Inferior, derecha, superior e izquierda.



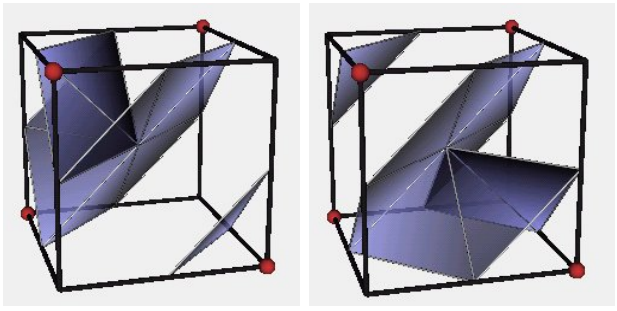
(a) Posición entrada: 1 (b) Posición nueva: 16

Figura 4.11: Triangulación 13C. Caras Separadas: Inferior, derecha, superior e izquierda.



(a) Posición entrada: 1 (b) Posición nueva: 15

Figura 4.12: Triangulación 13D. Caras Separadas: Frente y derecha.



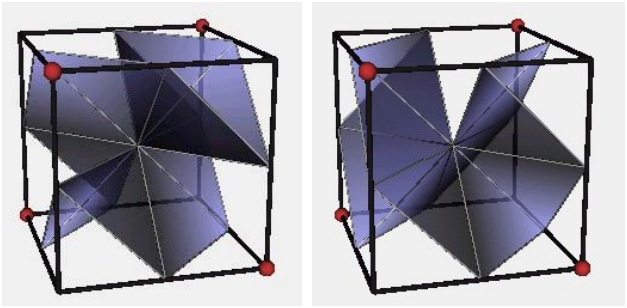
(a) Posición entrada: 2 (b) Posición nueva: 1

Figura 4.13: Triangulación 13D. Caras Separadas: Frente y derecha.

- Cuando son tres las caras separadas, se verifica si dos de ellas son opuestas. De ser este el caso, se determina con la función *"EsTreceE"* [3] si estamos ante un

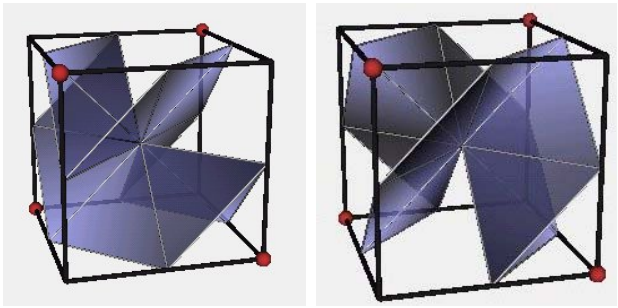
[3]EsTreceE: Esta función determina si se debe elegir la configuración 13E o 13F a partir del tipo de conexiones de sus caras.

caso 13E o 13F. En ambos casos es necesario modificar la posición usando un diccionario propio de cada triangulación (ver Cuadros 10.10,10.11 y Figuras 4.14,4.15).



(a) Posición entrada: 2 (b) Posición nueva: 19

Figura 4.14: Triangulación 13E. Caras Separadas: Inferior, derecha e izquierda.

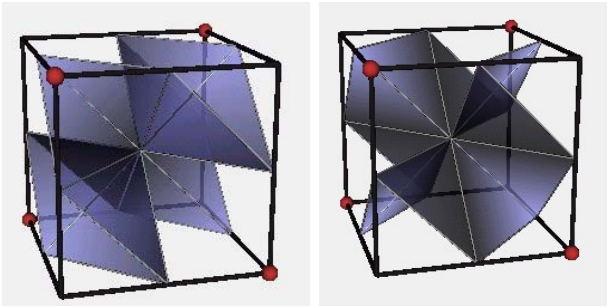


(a) Posición entrada: 2 (b) Posición nueva: 17

Figura 4.15: Triangulación 13F. Caras Separadas: Inferior, frente y fondo.

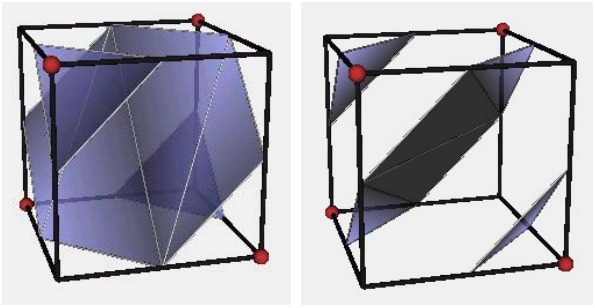
- Cuando dos de las tres caras no son opuestas, quiere decir que las tres concurren a un vértice del cubo. Por medio de la función *"BuscarPuntoConcurrente"*[4] obtenemos tal punto. Si este punto no está marcado, se elige la configuración 13G, en el caso contrario se elige la 13H y se invierten las normales si la posición de análisis es la 2. En ambos casos es necesario modificar la posición usando un diccionario propio de cada triangulación (ver Cuadros 10.12, 10.13 y Figuras 4.16,4.17).

[4]BuscarPuntoConcurrente: Esta función determina, a partir de tres caras cuál es el vértice común a todas.



(a) Posición entrada: 2 (b) Posición nueva: 19

Figura 4.16: Triangulación 13G. Caras Separadas: Inferior, fondo y derecha.



(a) Posición entrada: 2 (b) Posición nueva: 3

Figura 4.17: Triangulación 13H. Caras Separadas: Inferior, frente y derecha.

Una vez terminada la función, obtenemos la triangulación a usar, qué rotación debemos aplicarle y la orientación de sus normales. La triangulación la buscamos en la tabla de triangulaciones que nos indica las aristas en que se encuentran los vértices de los triángulos. Para determinar el lugar exacto de la arista donde colocar el vértice, interpolamos los valores de los extremos de la arista teniendo en cuenta el umbral usado. Luego debemos rotar estas aristas, por lo tanto, usando el índice de rotación y la tabla de rotaciones generamos la triangulación rotada. También, de ser necesario invertimos las normales de los mismos, con lo cual basta con reordenar sus vértices.

Una vez que determinamos los triángulos necesarios para generar la superficie que corta la celda en análisis, debemos generar una salida de tipo *vtkPolyData*. Para satisfacer este requerimiento almacenamos los triángulos en un objeto de la clase *vtkCellArray*; luego usando el método *void vtkPolyData::SetPolys (vtkCellArray *p)* es posible devolver un objeto de la clase *vtkPolyData*.

Capítulo 5

Verificación de correctitud del algoritmo

Para analizar la correctitud del algoritmo, es decir, que no genere superficies con huecos, implementamos un algoritmo que verifica lo siguiente: que para todas las posibles asignaciones de valores escalares a los vértices de las celdas que forman la grilla del volumen de datos, si se aplica el algoritmo *marching cubes* que implementamos en esta tesis, sólo se generan superficies con triangulaciones correctas.

Llamamos superficies con triangulaciones correctas a las superficies que no tienen huecos y que las normales de los triángulos apuntan hacia los vértices marcados. Para asegurarnos de cubrir todos los casos analizamos cómo se comportan dos cubos adyacentes y separamos la comprobación en los tres casos que cubren el total de combinaciones posibles (ver Figuras 5.1, 5.2 y 5.3).

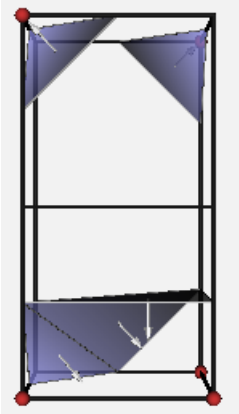


Figura 5.1: Los cubos compartiendo la cada derecha e izquierda.

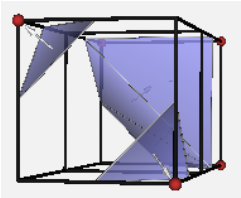


Figura 5.2: Los cubos compartiendo las caras de atrás y adelante.

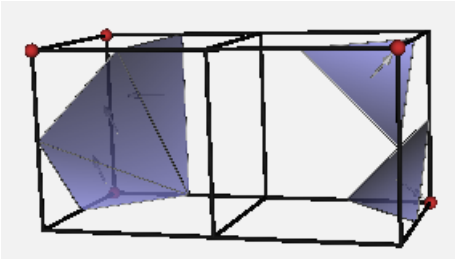


Figura 5.3: Los cubos compartiendo las caras superior y inferior.

En cada caso asignamos a los vértices de los cubos los posibles estados, al haber ocho vértices y dos estados se forman 256 casos. Pero si una cara tiene marcados sólo los vértices de una diagonal entonces esa cara es ambigua, por lo que también tenemos que tener en cuenta en la generación de los cubos que las caras pueden ser unidas o separadas.

Una vez asignados los valores a los vértices de los dos cubos verificamos que ambos tengan marcados los mismos vértices en la cara que comparten y ésta sea unida o separada para los dos cubos.

Luego generamos los triángulos utilizando el algoritmo *marching cubes* y verificamos que se forme una superficie continua en los triángulos que tienen vértices que pertenecen a la cara compartida por los dos cubos.

El sombreado de *Gouraud* y *Phong* pueden mejorar la apariencia del renderizado de triángulos. Ambas técnicas necesitan los vectores normales. Para aproximar la normal en los vertices de la superficie, es necesario que los puntos que delimitan los triángulos estén ordenados de forma consistente con las aristas de los triángulos vecinos (ver Figura 5.4). Esto significa que si el polígono p es definido por los puntos (1,2,3), entonces el triángulo adyacente que comparte la arista (2,3) debe usar la arista (2,3) en dirección (3,2). Si no es consistente entonces el promedio de las normales de los triángulos que comparten un punto puede ser cero o no representar la orientación de la superficie. Esto se debe a que la normal del triángulo se calcula haciendo un producto vectorial.

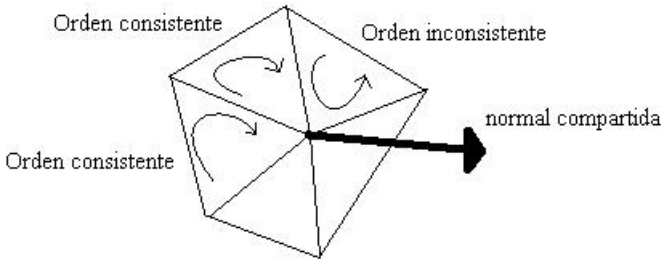


Figura 5.4: Triángulos definidos inconsistentemente.

Por lo anterior también verificamos que los triángulos estén definidos consistentemente.

Realizando estas verificaciones podemos asegurar que el algoritmo *marching cubes* implementado en este trabajo no tiene el problema de generación de huecos descripto en [Nie91].

Capítulo 6

The Visualization Toolkit (VTK)

6.1. Descripción

VTK es una biblioteca gratis *Open Source* que provee herramientas para el desarrollo de software de computación gráfica en 3D, procesamiento de imágenes y visualización científica. VTK incluye un libro de texto [Sch96], clases de C++, e interfaces para que la biblioteca pueda ser usada por otros lenguajes como Java, Tcl o Python. VTK puede ser usado en casi todas las implementaciones tipo UNIX y PC (Windows). El diseño de la biblioteca está muy influenciado por la programación orientada a objetos.

El modelo gráfico en VTK está en un nivel superior de abstracción que bibliotecas gráficas como OpenGL o PEX. Esto significa que es más fácil crear aplicaciones con utilidad práctica.

VTK es un verdadero sistema de visualización. No sólo permite visualizar la geometría, sino también incluye una gran cantidad de algoritmos de visualización como escalares, vectoriales, tensoriales, texturas, y volumétricos; y técnicas avanzadas como *implicit modeling, mesh smoothing, polygon reduction, cutting, contouring* y *Delaunay triangulation,* permitiéndonos integrar gran cantidad de algoritmos 2D y 3D para el desarrollo de software.

El objetivo de VTK es hacer que resulte fácil el desarrollo de software a estudiantes, académicos, desarrolladores de software o analizadores de datos.

6.2. Descripción técnica

6.2.1. Software

- Más de 700 clases C++, 350000 líneas de código C++.
- Diseñado utilizando las técnicas de Rumbaugh [Rum91].
- Más de 215000 líneas de código automáticamente generadas para *wrapper* con código TCL (similar cantidad para Python y Java) .
- Documentación en línea.
- Código C++ sencillo de comprender.

- Muchos ejemplos, aplicaciones, casos de test y datos para prueba.
- Soporta *multithreading* y algoritmos paralelos usando memoria distribuida.

6.2.2. Gráficos 3D

- Rendering 3D.
- Rendering de volúmenes (ray casting, soporte de hardware VolumePRO, posibilidad de mezclar superficies opacas y renderizado de volúmenes).
- Primitivas (puntos, líneas, polígonos, tira de triángulos, volúmenes).
- Visor y renderer interactivo.
- Luces (spot, infinita).
- Cámaras.
- Múltiples ventanas.
- Luces iluminan la escena.
- Cámaras definen el *viewport.*
- Actores especifican la geometría
- Posibilidad de armar jerarquías de actores.
- Visualización stereo.
- *Backface* y *Frontface culling* de polígonos.

6.2.3. Visualización

- Algoritmos escalares (mapeo de color, *marching cubes, dividing cubes, thresholding*).
- Algoritmos vectoriales (*hedgedhogs, streamlines, streampoints, streamsurfaces, streampolygon*).
- Algoritmos tensoriales (*tensor elipsoides, tensor glyphs, hyper-streamlines*).
- Algoritmos de modelado (esferas, conos, cilindros, cubos, líneas, planos).
- *Readers, Writes, Impoters* y *Exporters.*
- Tubería de procesos por demanda, con flujo de datos actualizado automáticamente.
- Conteo de referencias.
- Fuertemente tipado.

6.2.4. Filtros

- Difusores
- Pasa-bajos y pasa-altos.
- Convolución
- Diferencia, magnitud, aritméticos.
- Distancias.
- FFT.
- Histogramas.
- Cotas.

6.2.5. Modelo Gráfico

El modelo gráfico captura la esencia de un sistema gráfico de 3D de una forma que es fácil de comprender y usar.

Estos son los objetos básicos del modelo.

1. Render window - Uno o varios Renderers dibujan dentro de un Render Window para generar la escena.
2. Renderers - Coordinan el renderizado de las luces, cámaras y actores.
3. Luces - Iluminan los actores en escena.
4. Cámara - Contiene la posición de vista, el punto focal, y otras características de las cámaras.
5. Actor - Es un objeto dibujado por el renderer en la escena. Los actores son definidos con los mappers, propiedades, y transformaciones.
6. Propiedades - Representan los atributos de los actores. Los colores, tipo de iluminación (especular, difusa, ambiente) , texturas, el modo en que el actor es dibujado (sombreado o *wireframe*) y estilo de sombreado.
7. Mapper - Crea la geometría de los actores. Controla la creación de primitivas gráficas.
8. Transformaciones - Un objeto que consiste en una matriz de 4x4 que especifica la posición y orientación de los actores, cámaras y luces.

6.2.6. Modelo de Visualización

El modelo de VTK está basado en el paradigma *dataflow* adoptado por muchos sistemas comerciales. En este paradigma, módulos son conectados para formar una red. Los modulos realizan operaciones algorítmicas en los datos mientras estos transitan por la red. La ejecución de esta red de visualización es controlada en respuesta a demandas de datos (*demand-driven*) o en respuesta a eventos de los usuarios (*event-driven*). La ventaja de este modelo es que es flexible y rápidamente adaptable a diferentes tipos de datos o a nuevas implementaciones de algoritmos.

Construir redes de visualización es un proceso donde se conectan fuentes de datos y filtros. Lo importante es asegurarse que los *inputs* de un filtro son del tipo correcto, y que los ciclos de la red son administrados correctamente. Una vez que la red es construida se requiere un mecanismo para mantener actualizada la red con los datos de entrada o cuando los parámetros de los algoritmos cambian.

El tipado fuerte de C++ juega un rol importante. El método *SetInput()* de un filtro sólo acepta el tipo de dato especificado o sus subclases. Para conectar la salida del filtro A a la entrada del filtro B, una construcción semejante es utilizada: *B->SetInput(A->GetOutput())*; entonces la salida de A debe ser del mismo tipo que la entrada de B, o subclases del tipo de la entrada de B.

Capítulo 7

Comparación de resultados

7.1. Visualización

Usando el conjunto de datos *BuckyBall* (una molécula que está formada de 60 átomos de carbono estructurados en forma de domo geodésico) y el conjunto de datos *Headsq* (93 tomografías de un cráneo humano) mostramos los resultados de la implementación del algoritmo *marching cubes* con las configuraciones implementadas en este trabajo y las triangulaciones que usa la biblioteca VTK. VTK implementa una solución al problema de la generación de huecos agregando a las triangulaciones existentes los complementos de los casos que tienen caras ambiguas [Sch96]. De esta manera se asegura que no se produzcan las condiciones necesarias para la formación de huecos, pero no tiene en cuenta si la cara ambigua es unida o separada, por lo tanto la superficie generada no representa fielmente la realidad.

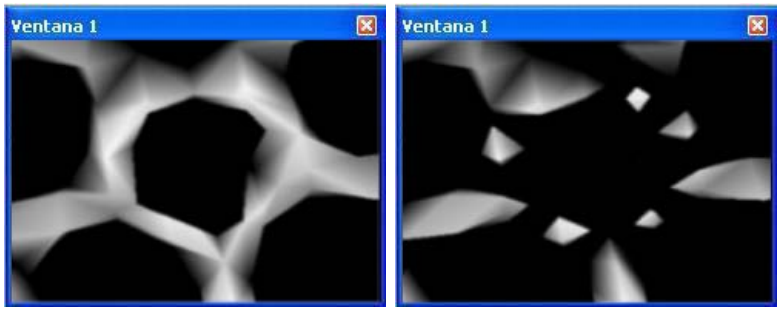
Se observa que usando las triangulaciones de VTK se obtienen superficies disconexas cuando no corresponde, situación que no ocurre utilizando la solución aquí propuesta

En las siguientes figuras, las imágenes de las derecha están construidas usando la versión de VTK del *marching cubes*, mientras que las imágenes de la izquierda se realizaron usando la versión implementada en este trabajo (ver Figura 7.1, 7.2, 7.5 y 7.6).

(a) Construido con vtkMarchingCubesTesis

(b) Construido con vtkMarchingCubes

Figura 7.1: Vista del conjunto de datos *BuckyBall*.



(a) Construido con vtkMarching-CubesTesis (b) Construido con vtkMarchingCubes

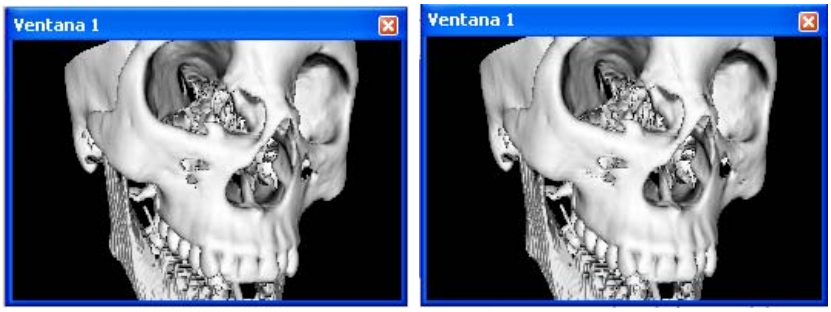
Figura 7.2: Ampliación de la zona problemática.

(a) Construido con vtkMarching-CubesTesis (b) Construido con vtkMarching-Cubes

Figura 7.3: Vista de la nuca del cráneo.

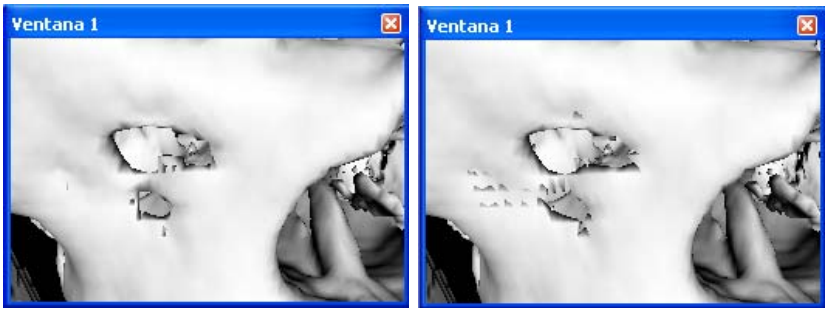
(a) Construido con vtkMarching-CubesTesis (b) Construido con vtkMarchingCubes

Figura 7.4: Ampliación de la zona problemática.



(a) Construido con vtkMarchingCubesTesis (b) Construido con vtkMarchingCubes

Figura 7.5: Vista del cráneo de frente.



(a) Construido con vtkMarchingCubesTesis (b) Construido con vtkMarchingCubes

Figura 7.6: Ampliación de la zona problemática.

7.2. Performance

Analizando los resultados obtenidos (ver Cuadro 7.1) podemos apreciar que tanto la cantidad de triángulos como el tiempo de procesamiento requerido por nuestro algoritmo es superior al implementado en la biblioteca VTK.

Algoritmo	Tamaño del volumen	Cantidad de triángulos	Tiempo (segundos)
Tesis	128x128x10	23382	0.88
Original VTK	128x128x10	23284	0.55
Tesis	128x128x50	146502	5.55
Original VTK	128x128x50	145030	3.85
Tesis	128x128x90	203096	8.08
Original VTK	128x128x90	201196	5.72

Cuadro 7.1: Comparación de performance.

Con respecto a la cantidad de triángulos, el hecho de contar con un punto intermedio en el *voxel* produce triangulaciones más precisas que requieren más triángulos.

Esto acarrea más tiempo de procesamiento, y además VKT, a diferencia nuestra, resuelve la triangulación a usar sólo accediendo a un arreglo sin realizar ningún otro tipo de análisis, lo cual explica la diferencia de tiempos.

Capítulo 8

Aplicación

8.1. Descripción

La forma en que se usa comúnmente VTK es realizando un programa que utiliza las funciones necesarias provistas por esta biblioteca con el objetivo de generar un resultado determinado. Como existen inconvenientes para los usuarios que no tienen conocimientos técnicos en programación, como ser médicos, matemáticos, físicos, etc., para realizar programas de visualización científica, desarrollamos una aplicación que permite usar de modo simple las funcionalidades que ofrece VTK.

Además, incorporamos a esta biblioteca la solución propuesta por Silvetti, logrando que esta nueva clase pueda ser utilizada de la misma forma en que se utiliza la implementación de *marching cubes* con que ya cuenta VTK.

Esta aplicación permite crear en forma interactiva fuentes de datos, ventanas, figuras, filtros, cámaras y relacionar todos éstos para renderizar una o varias figuras por separado o simultáneamente.

Cada objeto tiene sus propiedades que pueden ser modificadas en tiempo de ejecución, lo que permite una alta interactividad del usuario con la aplicación.

En cada ventana se pueden renderizar muchas figuras simultáneamente, en particular varias isosuperficies de un mismo conjunto de datos (ver Figura 8.1). Los datos de las figuras fueron tomados de tomografías computadas disponibles en la biblioteca VTK. Al poder asignarle a cada una de estas isosuperficies un valor de opacidad distinto, obtenemos resultados comparables con los obtenidos cuando se aplican métodos para renderizar volúmenes (*ray casting*, *cell splatting*). Sin embargo, el costo computacional del renderizado de superficies es notablemente inferior al del renderizado de volúmenes, ya que no es necesario volver a calcular la isosuperficie cada vez que se cambian sus propiedades (opacidad, color, posición, tipo de sombreado). En cambio, cuando se trabaja con renderizado directo de volúmenes cada vez que se modifica alguna propiedad del objeto es necesario volver a calcular su imagen, lo que trae aparejado un alto costo computacional, y permite una baja interactividad.

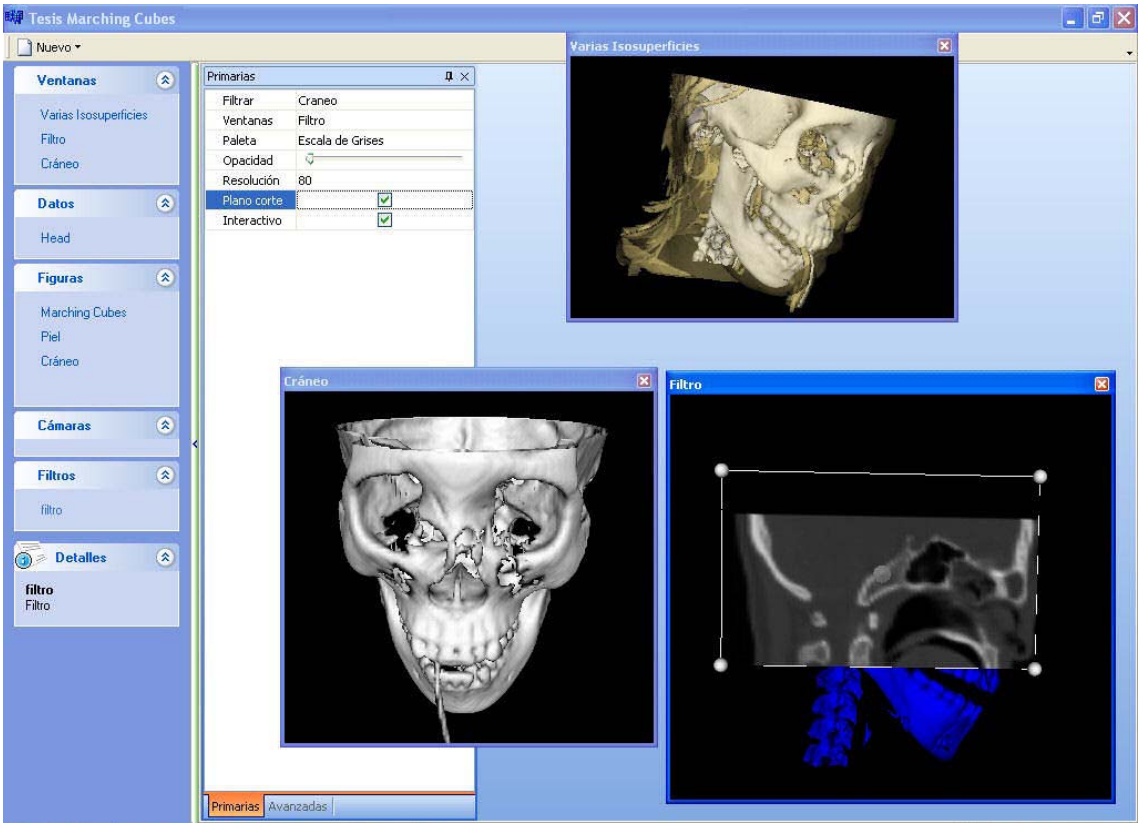


Figura 8.1: Aplicación.

Para visualizar un conjunto de datos determinado es práctico contar con varias vistas simultáneas del mismo. Si en nuestra aplicación se crea un conjunto de datos, una figura que use este conjunto y una ventana que lo muestre, luego podemos generar la cantidad de vistas deseadas con sólo crear ventanas que tengan la propiedad de renderizar las mismas figuras de alguna ventana ya existente (ver Figura 8.2).

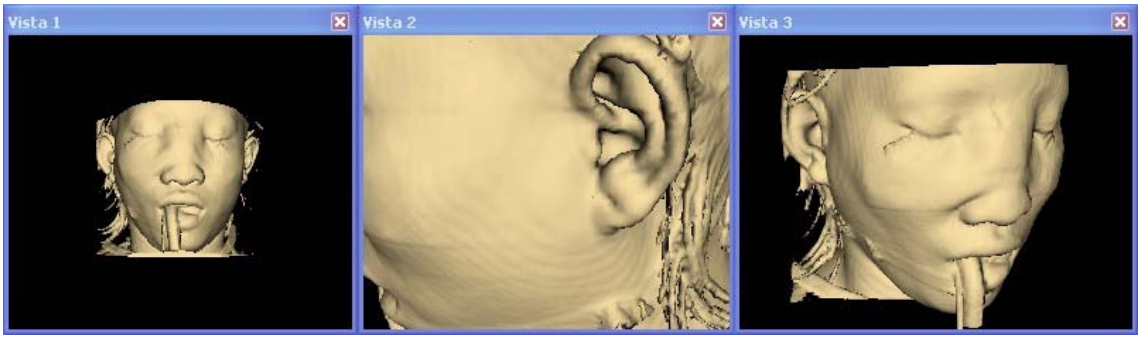


Figura 8.2: Varias vistas de un mismo conjunto de datos.

Otra opción es ver dos o más figuras distintas desde el mismo ángulo, en particular podemos querer ver dos isosuperficies del mismo conjunto de datos desde la misma posición. Esta funcionalidad se resuelve con el uso de cámaras. Como cada

ventana tiene una cámara asociada, podemos hacer que las ventanas que buscamos que tengan el mismo ángulo de visualización usen una misma cámara (ver Figura 8.3). Luego, la interacción del usuario con una ventana modifica la posición de la cámara asociada a la ventana, por lo tanto, todas las ventanas que tienen asociadas ésta cámara también se actualizan.

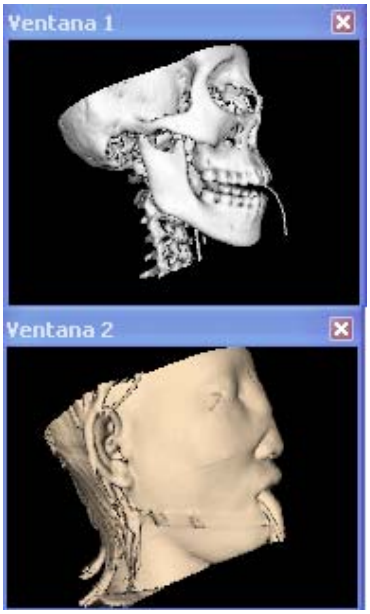


Figura 8.3: Misma cámara para distintas ventanas.

También incorporamos a la aplicación la posibilidad de cortar un volumen con un plano y visualizar la cara cortada del volumen con alguna paleta de colores generada por el usuario. Esto le permite al usuario, a partir de una serie de tomografías computadas o resonancias magnéticas, poder generar planos de corte en cualquier posición y/o en cualquier ángulo. Este plano es manejado interactivamente por el usuario con el *mouse* ya que cuenta con puntos en sus cuatro vértices que pueden ser arrastrados para modificar su tamaño y posición, también tiene un vector normal al plano y sólo basta moverlo para cambiar el ángulo de inclinación del plano (ver Figura 8.4).

La posibilidad de elegir la paleta con que se renderiza la imagen permite destacar órganos, huesos o zonas de interés.

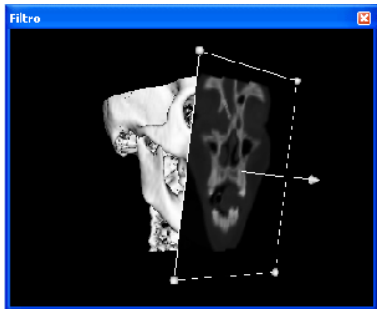


Figura 8.4: Plano de corte sobre un volumen.

Otra caracteristica con la que cuenta la aplicación es el uso de *wizards*. Estos sirven para generar figuras de forma sistemática, es decir, siguiendo una serie de pasos determinada, lo cual facilita la creación y vinculación de objetos.

Todas estas características permiten que usuarios sin conocimientos técnicos manejen la aplicación fácilmente.

8.2. Implementación

Contamos con un objeto de la clase *TPrincipal* que contiene un formulario, desde el cual se agregan o quitan los elementos deseados. Estos pueden ser: ventanas, conjuntos de datos, figuras, cámaras o filtros. Cada uno de éstos está controlado por una clase: *TManejadorVentanas*, *TManejadorSources*, *TManejadorFiguras*, *TManejadorCamaras* y *TManejadorFiltros*.

Por lo tanto, cada objeto de la clase *TPrincipal* cuenta con un objeto de cada una de las clases recientemente mencionadas. Esta clase interpreta los requerimientos que el usuario le solicita y en base a éstos deriva el pedido al objeto correspondiente.

Cada una de estas clases cuenta con estructuras de datos que le permiten almacenar un conjunto de elementos determinado (ver Cuadro 8.2) y formularios para mostrar y editar sus propiedades.

Clase	Repositorio de objetos de la/s clase/s
TManejadorVentanas	*TvtkBorlandRenderWindow.*
TManejadorSources	*vtkVolume16Reader* o *vtkImageReader* dependiendo del tipo de datos seleccionado.
TManejadorFiguras	*vtkMarchingCubesTesis*, *vtkMarchingCubes*, *vtkCubeSource*, *vtkPlaneSource*, *vtkSphereSource*, *vtkLineSource*, *vtkConeSource*, *vtkCylinderSource* dependiendo del tipo de figura seleccionada. También almacena un objeto de la clase *vtkLODActor* por cada uno de los objetos anteriormente mencionados que tiene almacenado.
TManejadorCamaras	*vtkCamera*
TManejadorFiltros	*TFiltro*
TFiltro	*vtkLODActor, vtkPlane, vtkPolyData, vtkProbeFilter, vtkPolyDataMapper, vtkPlaneWidget, vtkClipPolyData.*

Cuadro 8.2: Clases usadas en la aplicación.

Capítulo 9

Conclusiones y trabajo futuro

El algoritmo *marching cubes* es un excelente método para realizar la reconstrucción de superficies cuando se cuenta con datos volumétricos. Este algoritmo tiene múltiples aplicaciones, como por ejemplo en medicina, biología, matemática, física, etc. *Marching cubes* tiene la desventaja de necesitar una elevada cantidad de memoria para generar superficies con muchos detalles. La cantidad de memoria requerida puede disminuir si se representan los triángulos de tal forma que compartan aristas o vértices usando tecnicas de *triangle decimation* [Sch92].

La combinación de las funcionalidades provistas por la aplicación brinda al usuario la posibilidad de generar imágenes de gran calidad, así como también de analizarlas de la forma más conveniente. Por ejemplo, en la medicina puede ser usado para renderizar superficies a partir de tomografías computadas o resonancias magnéticas y ayudar a realizar diagnósticos con un alto grado de precisión.

La biblioteca VTK, al ser *open source*, se encuentra en continuo desarrollo. Esta característica es tenida en cuenta, ya que el diseño de la aplicación es modularizado, permitiendo de esta forma incorporar sencillamente nuevas funcionalidades.

Sería interesante analizar como se comporta este algoritmo con otros conjuntos de datos provenientes de diversas fuentes como simulaciones matemáticas, físicas o biológicas. También consideramos pendiente realizar una interpolación de segundo orden (*spline*) con los puntos que intersectan la isosuperficie, o aprovechar el pipe implementado en las placas gráficas de última generación.

En cuanto a la performance, en este trabajo se utilizan estructuras de datos de propósito general. Sería interesante implementar estos tipos de datos teniendo en cuenta el uso y las necesidades de nuestro algoritmo. Otro punto a tener en cuenta para mejorar la aplicación es la interfaz con el usuario. Se buscó lograr una herramienta que le sea útil a la mayor cantidad de usuarios posible y que le brinde la posibilidad de ir construyendo paso a paso la superficie a renderizar, presentando una amplia gama de propiedades para cada objeto que deben ser configuradas. Por ejemplo, no es lo mismo lo que le interesa analizar de un determinado conjunto de datos a un cirujano plástico que a un traumatólogo o a un matemático. Esto puede consumirle mayor tiempo que el deseado o bien generar confusión a usuarios no acostumbrados a manejar sistemas complejos de visualización científica. Por lo tanto, crear perfiles de usuario y en base a los mismos generar configuraciones por defecto para aliviar el proceso de carga y configuración de los datos es una mejora a implementar en la aplicación.

Capítulo 10

Apéndice

10.1. Glosario

Puntos marcados: Decimos que un vértice de una celda está marcado si su valor es mayor o igual al umbral con el cual estoy generando la superficie.

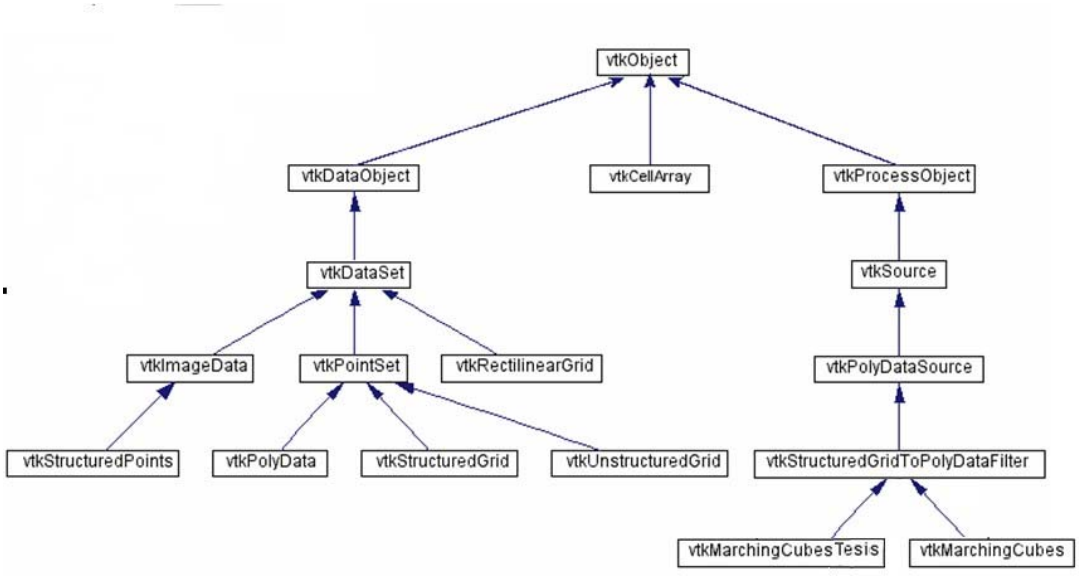


Figura 10.1: Estructura de clases usadas de VTK.

vtkStructuredPointsToPolyDataFilter: Es una clase abstracta que funciona como filtro, tomando como entrada un objeto un conjunto de puntos estructurados y genera data poligonal como salida.

vtkDataSet: Es una clase abstracta que especifica una interface para objetos de tipo conjunto de datos. También provee métodos que exponen características propias de la información que contiene.

vtkPolyData: Esta clase es una implementación concreta de *vtkDataSet* que representa una estructura geométrica formada por líneas, polígonos, vértices y tiras de triángulos.

vtkImageData: Esta clase es una implementación concreta de *vtkDataSet* que representa una estructura geométrica que es un arreglo de puntos topológica y geométricamente regular.

vtkCellArray: Es una lista de punteros a arreglos con la forma (cantPuntos1, pt11, pt12, .. pt1cantPuntos1, cantPuntos2, pt21, .. pt2cantPuntos2).

10.2. Tablas

10.2.1. Caso 7

Posición	**Cara Separada**	**Nueva posición**
1	Superior	1
1	Derecha	8
1	Frente	18
3	Superior	3
3	Izquierda	10
3	Frente	23
5	Superior	5
5	Izquierda	16
5	Fondo	20
7	Superior	7
7	Fondo	17
7	Derecha	2
9	Inferior	9
9	Derecha	4
9	Fondo	24
11	Inferior	11
11	Izquierda	14
11	Fondo	21
13	Inferior	13
13	Izquierda	12
13	Frente	22
15	Inferior	15
15	Derecha	6
15	Frente	19

Cuadro 10.1: Caso 7

10.2.2. Caso 10

Posición	Nueva posición	Nueva posición Complemento
1	13	7
6	12	10
7	15	1
10	6	16
18	21	24
24	18	21

Cuadro 10.2: Caso 10

10.2.3. Caso 12

Posición	Nueva posición
1	11
2	12
3	9
4	10
5	15
6	16
7	13
8	14
9	3
10	4
11	1
12	2
13	7
14	8
15	5
16	6
17	22
18	21
19	20
20	19
21	18
22	17
23	24
24	23

Cuadro 10.3: Caso 12

10.2.4. Caso 13B1

Superior	**Inferior**	**Fondo**	**Frente**	**Izquierda**	**Derecha**	**Nueva Posición**
Unida	Separada	Separada	Separada	Separada	Separada	2
Separada	Unida	Separada	Separada	Separada	Separada	6
Separada	Separada	Unida	Separada	Separada	Separada	11
Separada	Separada	Separada	Unida	Separada	Separada	15
Separada	Separada	Separada	Separada	Unida	Separada	21
Separada	Separada	Separada	Separada	Separada	Unida	17
Separada	Unida	Unida	Unida	Unida	Unida	2
Unida	Separada	Unida	Unida	Unida	Unida	6
Unida	Unida	Separada	Unida	Unida	Unida	11
Unida	Unida	Unida	Separada	Unida	Unida	15
Unida	Unida	Unida	Unida	Separada	Unida	21
Unida	Unida	Unida	Unida	Unida	Separada	17

Cuadro 10.4: Caso 13B1

10.2.5. Caso 13B2

Superior	**Inferior**	**Fondo**	**Frente**	**Izquierda**	**Derecha**	**Nueva Posición**
Unida	Separada	Separada	Separada	Separada	Separada	18
Separada	Unida	Separada	Separada	Separada	Separada	22
Separada	Separada	Unida	Separada	Separada	Separada	16
Separada	Separada	Separada	Unida	Separada	Separada	12
Separada	Separada	Separada	Separada	Unida	Separada	5
Separada	Separada	Separada	Separada	Separada	Unida	1
Separada	Unida	Unida	Unida	Unida	Unida	18
Unida	Separada	Unida	Unida	Unida	Unida	22
Unida	Unida	Separada	Unida	Unida	Unida	16
Unida	Unida	Unida	Separada	Unida	Unida	12
Unida	Unida	Unida	Unida	Separada	Unida	5
Unida	Unida	Unida	Unida	Unida	Separada	1

Cuadro 10.5: Caso 13B2

10.2.6. Caso 13C1

Superior	**Inferior**	**Fondo**	**Frente**	**Izquierda**	**Derecha**	**Nueva Posición**
Separada	Separada	Separada	Separada	Unida	Unida	19
Separada	Separada	Unida	Unida	Separada	Separada	11
Unida	Unida	Separada	Separada	Separada	Separada	14
Unida	Unida	Unida	Unida	Separada	Separada	19
Unida	Unida	Separada	Separada	Unida	Unida	11
Separada	Separada	Unida	Unida	Unida	Unida	14

Cuadro 10.6: Caso 13C1

10.2.7. Caso 13C2

Superior	**Inferior**	**Fondo**	**Frente**	**Izquierda**	**Derecha**	**Nueva Posición**
Separada	Separada	Separada	Separada	Unida	Unida	1
Separada	Separada	Unida	Unida	Separada	Separada	16
Unida	Unida	Separada	Separada	Separada	Separada	24
Unida	Unida	Unida	Unida	Separada	Separada	1
Unida	Unida	Separada	Separada	Unida	Unida	16
Separada	Separada	Unida	Unida	Unida	Unida	24

Cuadro 10.7: Caso 13C2

10.2.8. Caso 13D1

Superior	**Inferior**	**Fondo**	**Frente**	**Izquierda**	**Derecha**	**Nueva Posición**
Separada	Separada	Unida	Separada	Separada	Unida	7
Separada	Separada	Separada	Unida	Separada	Unida	15
Separada	Unida	Separada	Unida	Separada	Separada	6
Unida	Separada	Separada	Unida	Separada	Separada	10
Separada	Separada	Unida	Separada	Unida	Separada	11
Separada	Separada	Separada	Unida	Unida	Separada	3
Unida	Separada	Unida	Separada	Separada	Separada	2
Separada	Unida	Unida	Separada	Separada	Separada	14
Separada	Unida	Separada	Separada	Separada	Unida	19
Separada	Unida	Separada	Separada	Unida	Separada	21
Unida	Separada	Separada	Separada	Separada	Unida	17
Unida	Separada	Separada	Separada	Unida	Separada	23
Unida	Unida	Separada	Unida	Unida	Separada	7
Unida	Unida	Unida	Separada	Unida	Separada	15
Unida	Separada	Unida	Separada	Unida	Unida	6
Separada	Unida	Unida	Separada	Unida	Unida	10
Unida	Unida	Separada	Unida	Separada	Unida	11
Unida	Unida	Unida	Separada	Separada	Unida	3
Separada	Unida	Separada	Unida	Unida	Unida	2
Unida	Separada	Separada	Unida	Unida	Unida	14
Unida	Separada	Unida	Unida	Unida	Separada	19
Unida	Separada	Unida	Unida	Separada	Unida	21
Separada	Unida	Unida	Unida	Unida	Separada	17
Separada	Unida	Unida	Unida	Separada	Unida	23

Cuadro 10.8: Caso 13D1

10.2.9. Caso 13D2

Superior	**Inferior**	**Fondo**	**Frente**	**Izquierda**	**Derecha**	**Nueva Posición**
Separada	Separada	Unida	Separada	Separada	Unida	9
Separada	Separada	Separada	Unida	Separada	Unida	1
Separada	Unida	Separada	Unida	Separada	Separada	12
Unida	Separada	Separada	Unida	Separada	Separada	8
Separada	Separada	Unida	Separada	Unida	Separada	5
Separada	Separada	Separada	Unida	Unida	Separada	13
Unida	Separada	Unida	Separada	Separada	Separada	16
Separada	Unida	Unida	Separada	Separada	Separada	4
Separada	Unida	Separada	Separada	Separada	Unida	24
Separada	Unida	Separada	Separada	Unida	Separada	22
Unida	Separada	Separada	Separada	Separada	Unida	18
Unida	Separada	Separada	Separada	Unida	Separada	20
Unida	Unida	Separada	Unida	Unida	Separada	9
Unida	Unida	Unida	Separada	Unida	Separada	1
Unida	Separada	Unida	Separada	Unida	Unida	12
Separada	Unida	Unida	Separada	Unida	Unida	8
Unida	Unida	Separada	Unida	Separada	Unida	5
Unida	Unida	Unida	Separada	Separada	Unida	13
Separada	Unida	Separada	Unida	Unida	Unida	16
Unida	Separada	Separada	Unida	Unida	Unida	4
Unida	Separada	Unida	Unida	Unida	Separada	24
Unida	Separada	Unida	Unida	Separada	Unida	22
Separada	Unida	Unida	Unida	Unida	Separada	18
Separada	Unida	Unida	Unida	Separada	Unida	20

Cuadro 10.9: Caso 13D2

10.2.10. Caso 13E

Superior	**Inferior**	**Fondo**	**Frente**	**Izquierda**	**Derecha**	**Nueva Posición**
Unida	Unida	Separada	Separada	Separada	Unida	3
Unida	Unida	Separada	Separada	Unida	Separada	7
Unida	Unida	Separada	Unida	Separada	Separada	5
Unida	Separada	Unida	Unida	Separada	Separada	19
Unida	Unida	Unida	Separada	Separada	Separada	1
Separada	Separada	Separada	Unida	Unida	Unida	2
Separada	Separada	Unida	Separada	Unida	Unida	10
Separada	Separada	Unida	Unida	Separada	Unida	20
Separada	Separada	Unida	Unida	Unida	Separada	18
Separada	Unida	Separada	Separada	Unida	Unida	8
Separada	Unida	Unida	Unida	Separada	Separada	17
Unida	Separada	Separada	Separada	Unida	Unida	4

Cuadro 10.10: Caso 13E

10.2.11. Caso 13F

Superior	**Inferior**	**Fondo**	**Frente**	**Izquierda**	**Derecha**	**Nueva Posición**
Unida	Unida	Separada	Separada	Separada	Unida	18
Unida	Unida	Separada	Separada	Unida	Separada	22
Unida	Unida	Separada	Unida	Separada	Separada	10
Unida	Unida	Unida	Separada	Separada	Separada	14
Separada	Separada	Separada	Unida	Unida	Unida	1
Separada	Separada	Unida	Separada	Unida	Unida	5
Separada	Unida	Separada	Separada	Unida	Unida	19
Unida	Separada	Separada	Separada	Unida	Unida	17
Separada	Separada	Unida	Unida	Separada	Unida	7
Separada	Separada	Unida	Unida	Unida	Separada	3
Separada	Unida	Unida	Unida	Separada	Separada	12
Unida	Separada	Unida	Unida	Separada	Separada	8

Cuadro 10.11: Caso 13F

10.2.12. Caso 13G

Superior	Inferior	Fondo	Frente	Izquierda	Derecha	Nueva Posición
Unida	Separada	Unida	Separada	Unida	Separada	1
Unida	Separada	Unida	Separada	Separada	Unida	21
Unida	Separada	Separada	Unida	Unida	Separada	19
Unida	Separada	Separada	Unida	Separada	Unida	5
Separada	Unida	Unida	Separada	Unida	Separada	15
Separada	Unida	Unida	Separada	Separada	Unida	13
Separada	Unida	Separada	Unida	Unida	Separada	18
Separada	Unida	Separada	Unida	Separada	Unida	11

Cuadro 10.12: Caso 13G

10.2.13. Caso 13H

Superior	Inferior	Fondo	Frente	Izquierda	Derecha	Nueva Posición
Unida	Separada	Separada	Unida	Unida	Separada	1
Unida	Separada	Unida	Separada	Separada	Unida	5
Unida	Separada	Unida	Separada	Unida	Separada	3
Unida	Separada	Separada	Unida	Separada	Unida	7
Separada	Unida	Unida	Separada	Unida	Separada	12
Separada	Unida	Unida	Separada	Separada	Unida	1
Separada	Unida	Separada	Unida	Unida	Separada	5
Separada	Unida	Separada	Unida	Separada	Unida	4

Cuadro 10.13: Caso 13H

10.2.14. Matriz de transformación de aristas según posición

Arista Posición	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	8	4	9	2	11	6	10	1	3	7	5	12
3	3	0	1	2	7	4	5	6	11	8	9	10	12
4	9	0	8	4	10	2	11	6	5	1	3	7	12
5	2	3	0	1	6	7	4	5	10	11	8	9	12
6	4	9	0	8	6	10	2	11	7	5	1	3	12
7	1	2	3	0	5	6	7	4	9	10	11	8	12
8	8	4	9	0	11	6	10	2	3	7	5	1	12
9	4	7	6	5	0	3	2	1	9	8	11	10	12
10	2	10	6	11	0	9	4	8	3	1	5	7	12
11	5	4	7	2	1	0	3	11	10	9	8	11	12
12	11	2	10	6	8	0	9	4	7	3	1	5	12
13	6	5	4	7	2	1	0	3	11	10	9	8	12
14	6	11	2	10	4	8	0	9	5	7	3	1	12
15	7	6	5	4	3	2	1	0	8	11	10	9	12
16	10	6	11	2	9	4	8	0	1	5	7	3	12
17	9	5	10	1	8	7	11	3	0	4	6	2	12
18	3	11	7	8	1	10	5	9	0	2	6	4	12
19	8	3	11	7	9	1	10	5	4	0	2	6	12
20	1	9	5	10	3	8	7	11	2	0	4	6	12
21	10	1	9	5	11	3	8	7	6	2	0	4	12
22	7	8	3	11	5	9	1	10	6	4	0	2	12
23	11	7	8	3	10	5	9	1	2	6	4	0	12
24	5	10	1	9	7	11	3	8	4	6	2	0	12

Cuadro 10.14: Transformación de aristas según posición

Bibliografía

[Dur88] Durst, M. Letters: Additional reference to "marching cubes". *Computer Graphics*, 22:72–73, April 1988.

[Fol90] Foley, J., Van Dam, A., Feiner, S., and Hughes, J. *Computer Graphics. Principles and Practice.* Addison-Wesley, Reading, Massachussetts, second edition, 1990.

[Gon96] González, R. and Woods, R. *Digital Image Processing.* Addison-Wesley, Wilmington, USA, 1996.

[Lor87] Lorensen, W. and Cline, H. A High Resolution 3D Surface Construction Algorithm. *ACM Computer Graphics*, 21(4):163–169, 1987.

[Nie91] Nielson, G. and Hamann, B. The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes. In *Proceedings of the IEEE '91 Visualization Conference*, pages 83–91. IEEE, 1991.

[Rum91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. Object-Oriented Modelling and Design. 1991.

[Sch92] Schroeder, W., Zarge, J., and W.Lorensen. *Decimation of Triangle Meshes.* Computer Graphics (SIGGRAPH 92), August 1992.

[Sch96] Schroeder, W., Martin, K., and Lorensen, B. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 3rd Edition.* Prentice Hall, 1996.

[Sil01] Silvetti, A. F. Visualización científica de volúmenes. *Tesis de Magister en Ciencias de la Computación*, 2001.

[Wat92] Watt, A. and Watt, M. *Advanced Animation and Rendering Techniques.* Addison-Wesley, London, 1992.